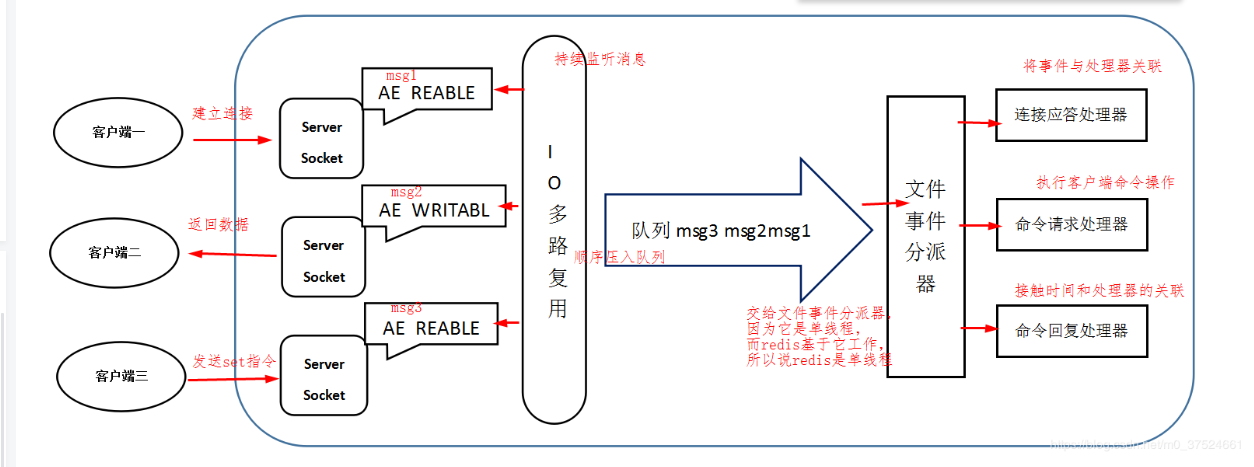
Redis为什么快

1. 基于内存的 读写速度快
2. 全程使用hash结构 读取速度快
3. 多路复用IO
   1. 多路指的是多个socket连接，复用指的是复用一个线程；可以让单个线程处理多个连接请求
   2. 一个线程通过记录跟踪多个IO流来管理多个IO流
   3. Select：一个女生下楼, select大妈都不知道这个是不是你的女神，她需要一个一 个 询问，并且select大妈能力还有限，最多一次帮 你监视1024个妹子
   4. Poll：poll大妈不限制盯着女生的数量，只要是经过宿舍楼门口的女生，都会帮你去问是不是你女神
   5. Epoll：epoll大妈不限制盯着女生的数量， epoll大妈会为每个进宿舍楼的女生脸上贴上一一个大字条,上面写上女生自己的名字,只要女姓下楼了, 　　eoll大妈就知道这个是不是你女神了,然后大妈再通知你.

Redis复制过程

Redis线程模型



文件事件处理器(单线程)

Redis过期策略

1. 惰性删除：使用的时候校验 是否过期 集群中从 怎么处理？
2. 定期删除：每隔100ms随机抽取 检查是否过期

Redis内存淘汰

1. allkeys-lru 内存不足时 删除最近最少使用
2. volatile-ttl 内存不足时 在设置了过期时间的key中删除更早过期的key

Redis击穿、穿透、雪崩

1. 击穿 Redis中失效 数据库中有数据 设置临时有效值 查库更新key
2. 穿透 Redis与数据库中都空 布隆过滤器 bitmap
3. 雪崩 同时过期大量数据或者机器宕机
   1. 高可用+随机时间
   2. 在系统中准备ehcache 存放热点
   3. 限流+降级 避免数据库被打死

Redis优化

1. 缩短键值对的存储长度
2. 设置过期时间
3. 禁用耗时长的命令 ：keys
4. 使用流水线
5. 避免缓存同时失效：Redis使用贪心策略 所以要规避Redis多次持续扫描
6. 使用lazyfree 惰性删除：在删除的时候提供异步延时释放键值的功能，把键值释放操作放在 BIO(Background I/O) 单独的子线程处理中，以减少删除删除对 Redis 主线程的阻塞

为什么要使用缓存？

1. 性能高
2. 并发高

缺点：

数据一致性

缓存穿透 击穿 雪崩等

Redis与memcached

1. 拥有更多的数据结构 支持更多场景
2. Redis有集群

Redis高并发 高可用 缓存一致性？

1. 支持10W+QPS: 读写分离 主从结构 redis集群 replication
2. 写多读少：异步
3. 高可用：全年99.99%的时间是可用状态
4. Redis高可用：哨兵做故障转移 主备切换

哨兵：(分布式的)

1. 集群监控：监控master和slave进程是否正常
2. 消息通知：redis某个实例故障 哨兵将消息发送给管理员
3. 故障转移：master挂掉了 会自动转移到slave上
4. 配置中心：故障转移发生后 通知client客户端新的master地址

哨兵作为一个集群去运行 互相协同工作

1. 故障转移时 判断master是否宕机 需要大部分哨兵同意才可以 分布式选举
2. 即使部分哨兵挂掉了 哨兵集群还是正常工作

哨兵核心知识：

1. 至少需要3个实例 保证自己的健壮性
2. 哨兵+redis的主从架构 保证不了数据零丢失 只能保证高可用

为什么哨兵集群需要2个以上实例？

1. Quorum(法定人数)的值：几个哨兵能判定故障(通常设置过半值)
2. 小于3个时 只要有一个挂掉 就不能正常的执行故障转移

Replication核心机制

1. 一个master node 可以配置多个slave node
2. Slave node 也可以连接其他的 slave node
3. Slave node 在做复制的时候 是不会 block master node ，也不会block自己的查询操作用旧的数据提供服务 复制完成的时候删除旧的数据 加载新的数据集 这个时候暂停服务
4. Slave node 主要用来进行横向扩容 做读写分离 提高吞吐率

Master持久化对主从架构的安全保障：(master 要做持久化 也要做数据备份)

Slave node做为master的数据热备，master宕机重启时 没有可恢复的数据 就会将空的数据同步到slave上面 这样数据就全部over了

复制的完整流程：

1、Slave在启动的时候就会保存master的IP以及host

2、Slave内部有个定时任务每秒都会check是否有master要连接 如果有就和master建立socket连接

3、slave向master发送命令

4、口令认证：如果master设置了requirepass 那么slave必须发送masterauth口令进行验证

5、master第一次会将全部数据发给slave

6、后续的数据异步发送给slave做到数据同步

数据同步相关的核心机制：(全量复制)

1. 通过offset比对就能知道数据的差异情况：master和slave都会维护一个offset，master与slave都会在自身不断累加offset，slave每秒都会上报自己的offset给master 同时master也会保存每个slave的offset
2. Backlog主要做增量复制：master有一个backlog 默认是1MB大小，master给slave复制数据时 也会将数据在backlog中同步写一份
3. Master run id :slave根据对runid的判断 得出是否做全量复制
4. Psync:slave节点使用psync从master进行复制 同时把runid offset 发送给master

Redis主从初始复制(master->slave)：

1. slave发送SYNC命令到master
2. master执行bgsave 生成快照文件(RDB)，发送给slave，如果RDB复制时间超过60s(repl-timeout)就会失败 同时开辟一个缓存区(client-output-buffer-limit slave 256MB 64MB 60 缓存区消耗持续超过64 或者一次性超过256 那么停止复制 复制失败) 存储当前发生的命令
3. slave根据配置决定是否使用旧数据提供服务
4. slave丢弃所有旧数据 开始加载master传的快照文件，完成解析 开始提供服务
5. master把缓存区中的命令同步给slave ，
6. 同步以后 每执行一个命令都同步给slave

增量复制：

1. 全量复制过程中 如果出现复制中断，那么slave重新连接master时 会触发增量复制
2. Master直接从自己的backlog中获取到offset 信息 从offset记录的节点 继续复制

主从复制 断点续传

1. master node 会在内存中维护一个backlog， master和slave 都会保存一个 replica offset 还有一个master ID，offset就是保存在backlog中的 如果master和slave网络连接断掉了 slave会让master从上次的replica offset开始继续复制，但是如果没有找到offset就会执行一次全复制

无磁盘化复制：

1. master在内存中直接创建RDB 然后发送给slave 不会落在本地磁盘中
2. repl-diskless-sync repl-diskless-sync-delay 等待时间(等待更多的slave连接)

过期KEY：slave是不会过期key的 master会把过期命令发送给slave

Heartbeat：主从心跳

Master->slave :每隔10秒发一次 slave->master:每隔1秒发送一次

主从架构：

1. master异步同步数据给slave时，出现异常 造成数据丢失
   1. 解决：min-slaves-to-write 1 min-slaves-max-lag 10 :要求至少有1个salve 数据复制和同步的延迟不能超过10秒，如果都超过了10秒 master就不会再接收任何请求(通常情况会在client做降级 将数据写到本地磁盘 client对外接收的请求 再做降级 做限流 减慢请求涌入速度，或者临时灌入一个MQ队列)
2. 集群脑裂导致数据丢失：master出现了异常性的 有相同数据 相同工作的两个节点，当脑裂恢复时 原master数据会被新master覆盖 造成数据丢失(集群脑裂通常是发生在集群中部分节点之间不可达而引起的 或者因为节点请求压力较大 导致其他节点与该节点心跳检测不可用)
   1. 解决：min-slaves-to-write 1 min-slaves-max-lag 10

哨兵：

1. sdown和odown转换机制
   1. sdown:主观宕机 任何一个哨兵觉得master宕了 就是主观宕机
   2. odown:客观宕机 quorum数量的哨兵都觉得master宕机了那么就是客观宕机
   3. sdown与odown之间的转换依据就是：是否达到quorum值
2. 哨兵和slave集群的自动发现机制
   1. 哨兵互相之间的发现是通过redis的pub/sub系统实现的，每个哨兵都会往\_sentinel\_:hello 的channel里发一个消息 这时候其他哨兵都可以消费这个消息，从而感知到其他哨兵的存在
   2. 每隔两秒 每个哨兵会给自己监控的实例对应的sentinel：hello channel 发送消息 内容是自己的host IP 和runid 还有对这个master的监控配置
   3. 每个哨兵还会和其他哨兵交换对master的监控配置 互相进行监控配置的同步
3. 选举算法
   1. 如果master被认为是odown了 而且majority哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先需要选举一个slave
   2. 跟master断开连接的时长:如果一个salve跟master断开连接的时间超过了down-after-milliseconds的10倍 外加master宕机的时长 那么就被认为不适合选举为master
   3. Salve优先级：根据salve priority 配置排序 越低优先级越高
   4. 复制offset：offset越靠后 优先级越高(越靠后，说明与原master同步的数据越多，所以优先级就高)
   5. Run id ：runid小的优先级高
4. Quorum 与majority

每次在做主备切换时 首先需要quorum的哨兵确认是odown 同时需要majority的哨兵授权切换 才能开始切换(注意：这里的授权个数是取quorum与majority的最大值)

1. Configuration传播

哨兵完成切换后 会更新生成的master最新配置 然后通过pub/sub同步给其他哨兵，每个master的配置 都会对应新的version编号，slave根据version来确认是否更新master配置

redis持久化：(意义：故障恢复)

1. RDB(redis 默认开启，直接存储数据文件)
   1. 对redis数据执行周期性的持久化，适合做冷备数据
   2. 可以让redis保持高性能，因为只需要fork一个子进程
   3. 基于RDB文件做重启和恢复 更加快捷
   4. 两次生成RDB的间隔时间内数据可能丢失，间隔时间太长数据量很大时 会造成redis服务性能损耗
2. AOF(都开启的情况下默认优先使用AOF，存储命令)
   1. 对每条写入命令作为日志，以append-only的模式写入一个日志文件中，没有任何磁盘寻址的开销，写入性能高 文件不易破损
   2. 现在的操作系统 写文件不会直接操作磁盘 会先写cache 然后一定的时间从cache到diskfile,redis 中每隔1秒调用一次操作系统的fsync命令 将数据从cache中刷到磁盘文件中
   3. AOF文件会越来越大 当到了一定的值时 AOF会做rewrite操作(基于当时redis中的数据 构造一个新的AOF 把就得删除)
   4. 可以更好的保护数据不丢失
   5. 同样的数据 比RDB的文件大
3. 综合使用AOF+RDB AOF做数据恢复的第一选择，RDB做不同程度的冷备 在AOF文件丢失或者损坏的情况下 使用RDB恢复数据

Redis集群原理

Redis-cluster

Redis-sharding 客户端分区方案:客户端就采用hash算法决定key存到哪个节点上

分区逻辑可控 配置简单 节点间无关联 容易扩展

客户端无法动态增删服务节点，需要自行维护分发逻辑，客户端之间无连接共享。

Redis-cluster(多master+读写分离+高可用 适用于海量数据)

支撑N个master node 每个master可以有多个slave，

查询路由方案：客户端随机的请求一个redis实例，然后由Redis转发给正确的Redis节点，在客户端的帮助下直接重定向到正确的Redis节点

无中心节点 按照数据槽存储在多个实例上 可以平滑的扩容 支持高可用和故障转移

故障转移节点检测没有zk及时

数据分区

1. 节点取余分区 hash key值 取余
   1. 简单、扩容时采用翻倍扩容 避免了全量迁移的发生
   2. 节点变化时会导致数据迁移的发生
2. 一致性哈希分区 将存储节点首尾相连的排列在hash环上
   1. 数据变更时只影响后续节点
   2. 加减节点会造成环中部分数据无法命中
3. 虚拟槽分区
   1. 使用哈希函数把数据映射到一个固定范围的整数集合中(槽0-16383)，槽是数据操作的基本单位 slot=CRC16(key)&16383
   2. 增加和删除节点时 虚拟槽会重新分配保持总体不变
   3. 解耦数据和节点之间的关系 更容易扩容收缩，节点自身维护槽的映射关系 不需要客户端或者代理服务维护。
4. 集群一般由多个节点组成 至少6个 三主三从 都需要配置cluster-enabled yes

Redis cluster

自动将数据进行分片，每个master上放一部分数据，提供内置的高可用支持，部分master不可用时 依然可以工作

在cluster的架构下 每个redis放开两个端口号 比如一个是6379 另一个就是16379(加10000) 16379是用来进行节点间通信的 每个节点每隔一段时间都会向别的节点发送ping消息 用来进行故障检测 配置更新 故障转移授权

数据分布的算法

1. 一致性hash算法
   1. 圆环 根据计算的hash值顺时针旋转 找到最近的节点
   2. 某个master异常 只会影响到自己 不会影响到别的master上的数据
   3. 虽然避免了整体master不可用 但是会出现master热点问题
   4. 通过做虚拟节点 解决热点问题
2. Redis cluster的Hash slot
   1. Redis cluster有16384个虚拟卡槽 key计算CRC16的值 然后对16384取模 得到对应的卡槽值(数据分布针对的是卡槽不在是redis实例)
   2. 增加和删除master都会对当前生效的master重新分配卡槽，扩容简便，故障影响减小

Cluster的原理

1. 通信原理
   1. 节点间采取gossip协议(小道留言)
      1. 每个master都会维护完整的集群元数据信息，当某个master发生变化时 就把变更发送给别的master
      2. Gossip与集中式对比
         1. 集中式(zk)：元数据的操作时效性比较好 但是多有的元数据压力都集中在一个地方 造成存储压力
         2. Gossip:元数据的更新是分散的 有一定的延时，
   2. 集群中有很多的元数据 包括：hashslot->node 之间的映射关系，还有master->slave之间的关系 故障信息等
2. Gossip协议
   1. <Meet:某个节点> 发送meet给新加入的节点 让新节点加入集群中 然后新节点就会开始与其他节点进行通信
   2. Ping：每个节点都会频繁给其他节点发送ping 其中包含自己的状态还有自己维护的集群元数据 互相通过ping交换元数据
      1. 每个节点每秒会执行10次ping，每次会选择5个最久没有通信的其他节点，如果发现某个节点通信延时达到了cluster\_mode\_timeout/2 那么立即发送ping 避免数据交换延时过长，cluster\_node\_timeout可以调节 如果调节比较大 那么就会降低发送的频率，每次ping 一个是带上自己节点的信息 还有就是带上整个集群1/10其他节点的信息 进行数据交换
   3. Pong：返回ping和meet 包含自己的状态和其他信息 也可以用于信息广播和更新
   4. Fail：某个节点判断另一个节点fail之后就会通知其他节点 指定的节点宕机了
3. 面向集群的jedis内部实现原理
   1. 请求重定向
   2. Hash tag 可以指定同一个slot
   3. 重定向很消耗网络IO 所以大部分客户端 都是smart的：本地维护一份 hashslot->node的映射表 缓存
   4. Jediscluster
      1. 在jediscluster初始化的时候 随机选择一个node 初始化hashslot->node映射表 同时为每个节点创建一个JedisPool连接池，每次基于jediscluster执行操作 首先都会在本地计算key的hashslot 然后在本地映射表中找到对应节点，如果发现对应节点返回moved 那么利用该节点的元数据 更新映射表 重复上面的步骤知道找到对应节点 超过5次就会报错
   5. Hashslot迁移和ask重定向
      1. 如果正在迁移 那么会返回ask重定向给jedis，jedis接收到ask重定向之后 会重新定位到目标节点去执行，但是因为ask发生在卡槽迁移过程中 所以不会更新本地hashslot缓存
4. 高可用与主备切换原理(与哨兵类似)
   1. 判断节点宕机
      1. 一个节点认为宕机pfail 主观宕机
      2. 多个节点都认为宕机fail 客观宕机
      3. 在cluster-node-timeout内 某个节点一直没有返回pong 那么就认为是pfail
      4. 如果一个节点认为目标节点pfail了 就会在gossip消息中 ping给其他节点 如果超过半数的认为pfail了 就会变成fail
   2. 从节点过滤
      1. 对宕机的master 从他所有的slave中 选择一个切换成master
      2. 检查每个slave与masternode 断开连接的时间 如果超过了cluster-node-timeout \* cluster-slave-validity-factor 那么就没有资格切换成master
   3. 从节点选举
      1. 根据自己的offset 设置一个选举时间 offset越大 越靠前
      2. 所有的master给slave投票 过半则选举通过
   4. Cluster集成了哨兵与replication的机制
5. Cache Aside Pattern :读的时候 先读缓存 再读数据库，更新的时候 先删除缓存 再操作数据库
6. 数据库与缓存数据不一致问题
   1. 先改数据库 再删缓存 操作缓存时失败
      1. 先操作缓存 在操作数据库
   2. 更新数据时 同时读取 并发
      1. 数据库与缓存的更新、读取操作进行异步串行化： 相同的ID，进行hash取值然后对内存队列进行取模 这样每个商品就路由到了一个内存队列中(操作数据库) 也就有了先后顺序
      2. 分布式部署的情况下 怎么保证同一个请求 一直分发到一个服务器上？
         1. 按照某个参数hash路由
7. Redis CAS方案解决并发竞争(多个实例操作同一个资源)\_
   1. 分布式锁:确保同一时间只能有一个实例操作同一个key
   2. 每次在操作key之前 判断一下这个操作的时间是否比该key上一个操作新
8. 公司的redis集群部署
   1. Jimdb相关性能与架构
   2. 